VERIFYING FORMAL SPECIFICATIONS OF SYNCHRONOUS PROCESSES*

Patricia P. Griffiths[+] and Charles J. Prenner[++]

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts 02138

ABSTRACT: SYNVER is an automatic programming system for the synthesis of solutions to problems of synchronization among concurrent processes from specifications written in a high level assertion language (SAL). The correctness of the solutions constructed by SYNVER follows from the soundness of the synthesizer itself and from a verification phase which is applied to the specifications. This verification phase is the main topic of this paper. To provide context for the verification the paper includes a discussion of synchronization problems and a brief overview of both the SYNVER system and the SAL specification language. A formal definition of the correctness of a SAL specification is then presented along with algorithms which may be used to determine if a given specification is correct.

1. INTRODUCTION

This paper describes some of our recent work in the area of automatic programming. While this area has been defined in many ways, we view it as a collection of tools, methods, and techniques for fast, inexpensive production of reliable software. Automatic programming attempts to put more of the burden of software production and reliability upon the machine or software production system, and less upon the programmer or user. These motivations are similar to those that led to the development of high level programming languages as a replacement for assembly languages. Then, as now, the desire is to produce software which does as much work as possible for the programmer.

We are specifically concerned with developing approaches and techniques that advance toward the long-range goal of constructing general purpose automatic programming systems. Here, we begin work on some prerequisites of this goal through the construction of such a system for a specific problem domain. The value of this effort is twofold. In addition to examining outlines and directions for eventual more general systems, we have constructed an operational system for a non-trivial problem domain. Thus, we present an existence proof that many of the problems can be simplified and clarified by a judicious choice of problem domain and by careful system design.

The problem domain chosen is the synchronization of systems of concurrent processes. Conventional approaches have made the description and solution of such problems quite difficult. Our system, SYNVER, overcomes these difficulties by approaching synchronization at a level close to a human conceptual model of such problems.

The design and construction of the SYNVER system [Grif74,Grif75a] involves three areas of automatic programming: very high level languages, program synthesis, and program verification.

Traditionally, the main efforts in automatic programming have been in the area of verification. Automatic or semi-automatic theorem-provers formulate and prove theorems about logical formulae which are assertions or verification conditions about programs. From the resulting proofs, programs can be extracted to satisfy the theorems proven. Thus, synthesis of programs was originally based on verification.

SYNVER approaches synthesis and verification as separate issues. A very high level specification language (SAL) is formulated in which synchronization problems can be described in a high-level, formal, yet human-oriented way. (SAL is an extension of the extensible language ECL [Weg70][Weg74], using data, operator, and control extension facilities [Pren72]. SYNVER is itself written in ECL as well.) It is

* This research was supported in part by the Advanced Research Projects Agency under contract F19628-74-C-0083 and by IBM under an IBM fellowship grant.

+ Current Address: IBM Research Laboratory, San Jose, California 95193

++ Current Address: Computer Science Division, University of California at Berkeley, Berkeley, California 94720

this specification language together with problem-domain-specific knowledge which provides sufficient information to facilitate the synthesis and verification of solutions to the described problems. The synthesis and verification can be done independently; synthesis, therefore, is freed from its dependence on verification and from the limitations of theorem provers. In this paper we will be concerned primarily with the verification phase of SYNVER. However we must first provide both context and motivation for the verification techniques. In Sections 2 and 3 we discuss the nature of synchronization problems and previous techniques for their description and solution. Sections 4 and 5 give an overview of SYNVER and a brief discussion of the salient points of the specification language SAL. Sections 6 and 7 define specification verification. Section 8 discusses the algorithms used by the verification phase. Section 9 discusses extensions to these techniques.

2. SYNCHRONIZATION PROBLEMS

The basic unit of discourse in synchronization problems is the process. Wulf et al. [Wulf74] describe a process as the smallest entity that can be scheduled for independent execution. One can visualize a process as a program plus a data environment and a control environment together with enough status information (e.g., a program counter, a set of interrupts enabled or disabled, etc.) so that the process can be executed.

A processor is the active agent; it executes a process. Throughout this paper it is assumed that there exist a fixed number of processors available for the simultaneous evaluation of processes. The processors are multiplexed over the processes in order to insure that each process receives some share of processor time. However, no assumptions are made about the allocation of processor time. Because the multiplexing is transparent to the processes, in any given process an arbitrary amount of time may pass between the execution of one instruction and the next. Consequently, without some mechanism for coordination, communication between processes, such as sharing a resource, would be quite unreliable.

The possibility of interference between systems of concurrent processes characterizes what we call synchronization problems. Such systems are characterized by two or more processes, running concurrently on the same or different machines which share some data or resources. These processes are mainly asynchronous; usually they are operating independently on their own computations. Occasionally, however, they need to access shared data or resources or to make contact with each other. We can describe several typical kinds of synchronization problems:

1) Mutual exclusion problems. These problems occur when processes want exclusive access to a resource. Examples of such resources are line printers or disk files needed for output.

The concept of processes mutually excluding each other from resource access extends to program code as well. It is occasionally the case that processes have sections of code, called critical sections, which contain operations on shared data which are to be performed without interference from other processes. That is, these critical sections must be evaluated indivisibly with respect to other processes. Consequently, no more than one process at a time should be evaluating its critical section.

2) Shared access. Mutual exclusion problems generalize to requests for multiple, yet still restricted access. In this type of problem, resource access is not restricted to one process at a time. Instead, several processes of special types or in limited numbers may be permitted simultaneous access. An example of such a case is a file which can be accessed by any number of processes which access it on a read-only basis. However, processes which write on the file are still excluded.

3) Cooperation problems. Cooperation type synchronization problems occur when a process requires simultaneous access to two or more resources. These resources cannot in general be acquired one by one and kept until they are all accumulated because if the number of resources is limited, they may be exhausted before any process has filled its requirements. Thus all processes may be waiting for additional resources; none may be able to proceed. This situation is known as total deadlock [Holt71a, Holt 71b]. One way of avoiding this situation is for a process to acquire all its needed resources simultaneously, or else wait, retaining none of them.

Our knowledge of synchronization problems is quite informal. We know that processes generally acquire and release resources one by one, or if they acquire several at a time, they generally later release the same number. In a system of synchronous processes, processes generally come to some section of code which they must not enter before making some arrangements with other processes. Presumably this code uses scarce resources, and the arrangements are in the form of obtaining exclusive access, waiting for sufficient resources, and so on. At the end of this section of code, processes generally notify other processes that the restricted access has been lifted.

Before entering this critical section of code, a process either obtains access and proceeds, or waits until access can be obtained. A process in a synchronous system never interrupts another process and causes it to wait.

It is characteristic of synchronization problems that if a process can either wait or continue upon attempting to enter a critical section, then that waiting is transparent to the process. Once inside

the critical section, the state of the world from the process's viewpoint is the same whether it was delayed and then awakened or whether it proceeded directly. We call this commutativity the delay invisibility principle.

3. SOLVING SYNCHRONIZATION PROBLEMS

Many primitive operations have been invented to implement solutions to synchronization problems. One of the original synchronization primitives is Dijkstra's semaphore operations [Dij68], which we will consider as a representative example. Simply, a semaphore counts the scarce resource usage. Its initial value is the initial number of available resources. When its value is zero, no resources are available, and processes requesting resources have to wait. The P operation on a semaphore requests the resource, the V operation releases it.

Semaphore operations are ideal for describing very simple synchronization problems. They are not, however, easily adaptable to situations of shared data access where, for example, one type of process has priority over another [Cour71]. Nor are they easy to use for cooperation type synchronization problems [Par75]. Many of the more complex synchronization problems are of these latter two types. Semaphores and other similar types of synchronization primitives do not provide a natural model for human conceptualization of these kinds of problems.

It is our view that the human conceptual model of synchronization problems is centralized. A person tends to think not about each of the individual processes, but about what happens in the system of processes as a whole. The human problem conception focuses on how the scarce resource (be it critical section code, line printer, disk buffer, etc.) may be accessed. This encourages thinking about synchronization problems in terms of "states of the entire system" with respect to resource access.

Specification of synchronization problems in terms of their solutions with semaphore operations (or other low-level primitives) is inconsistent with such a centralized, state-oriented representation. Semaphores distribute the knowledge about synchronization among the various processes, while the human mind conceives of this knowledge in a centralized way. In addition, semaphores are very implementation-oriented. The gap between the conceptual model in terms of states and the solution in terms of P and V operations in each process is substantial. The problem solutions bear little resemblance to the problem description in English. Hence, in coding solutions to these problems, there is a difficult transformation from the human problem conception to the resultant code.

These implementation-oriented solutions are also difficult to change. If an incremental change is made in the problem description (such as giving one type of process priority but keeping the rest of the problem the same), there is often a vast change in the P and V solution to the problem. This does not reflect our intuition that the translation between mental conception and the machineable problem specification and solution should be a continuous function.

In addition, P and V solutions are cast at too low a level for a programmer to visualize all the possible interactions between processes. He has difficulty envisioning all the possible orderings of events. This makes it difficult to write programs to solve synchronization problems, and further, to prove that such a system of synchronous processes is correct.

The verification techniques we present are related to those of Robinson and Holt [Rob74] and Levitt [Lev72]. Their techniques in turn are based on work by Manna [Man69] [Man70] and Aschcroft and Manna [Ash70]. Manna formalizes properties of non-deterministic programs executed by a single processor. He introduces the choice point which corresponds to the set of instructions from which the processor may select at that point. Correctness of non-deterministic programs is related to the set of all possible executions of the program on a given input. Manna and Aschcroft formalize properties of parallel programs in a similar manner. A parallel program can be transformed into an equivalent non-deterministic program which, when executed, chooses one particular interleaving of instructions of the parallel programs from the set of all possible interleavings. Because of the proliferation of choice points, and thus the number of assertions as well, the concept of protected bodies of code (those containing no choice points and therefore executed indivisibly with respect to other processes) is introduced. Levitt extends the Aschcroft and Manna techniques. He retains the notion of protected bodies of code, and in addition to the choice node he introduces a split node to split a control path previously executed by one processor into two or more paths, each executed by its own processor. As with the techniques for sequential programs [Floyd67], Levitt divides control paths into fragments of code bounded by assertions and proves each piece separately. Thus, the synchronization may be proven correct in the same way as and at the same time as the asynchronous code. Robinson and Holt present a formal specification approach in which the code involved with synchronization is separated from the asynchronous programs. High level abstract entities (gates) control access to critical sections of code. Proving the synchronization correct in this context means proving that these gates protect critical sections in the desired manner with respect to global invariants. However, the verification technique is not easily mechanized. In addition, no implementation of gates is suggested.

4. OVERVIEW OF THE SYNVER SYSTEM

These difficulties have led us to propose an automatic programming approach to solving synchronization problems. Using this approach, a programmer can present his problem description in the centralized state-oriented way in which he conceives of it. He is provided with a very high level specification language SAL (Synchronization Assertion Language) which provides a natural medium for describing synchronization problems. Thus problem specifications can be easily read, modified, and communicated to others. This is not necessarily the case for semaphore solutions of synchronization problems.

Because SAL is a high level human-oriented medium for the description of synchronization problems, the choice of primitives in which to implement the specifications is separable from the problem itself. This enables such problems to be stated more clearly and frees the solutions from implementation-specific restrictions. The linguistic power of SAL allows incremental changes in the problem to be reflected by incremental, local changes to the SAL problem description. In addition, the specifications are formal enough to permit synthesis and verification of programs which solve the specified problem.

A SAL specification is in the form of an abstract program specification for each type of process in the system. This abstract program specification contains a skeleton of program code, including calls upon one or more synch functions. These synch functions, the code for which will be synthesized by SYNVER, provide the only means by which processes may communicate. Synch functions may be either monitor procedures [Hoare74], calls on Prenner's control interpreter [Pren73], or procedures utilizing Dijkstra's P and V operations [Steele75]. Both SAL and the verification technique are independent of the choice of synchronization primitives used, i.e., SYNVER can synthesize synch functions which utilize any of these various sets of lower level primitives. Synch function calls are indivisible with respect to other synch function calls, i.e., all such calls are ordered linearly in time. The code for the synch functions is completely independent of the code for the asynchronous computations of the processes. We believe that this separability is a desirable feature for communicating processes. It avoids "collusion" (unwitting or otherwise) arising from processes knowing too much about each other's internal computations [Byrn74]. This kind of separability and minimization of knowledge follows naturally from the principles of structured programming, where procedural encapsulation is intended to minimize the assumptions one component makes about another [Zilles73].

If we consider the system of processes to be in one of several distinct global states, and synch functions are the only means of communication between processes, then only execution of synch functions may cause a change of state. For synchronization problems there is usually only a small number of legitimate states that the set of communicating processes can be in. These global states are most often related to which processes are executing their critical sections.

SAL provides facilities for describing these global states and for specifying the transitions between these states made by the synch functions. The former is achieved using SAL state definition statements. The latter is achieved by placing assertions before and after each synch function call.

Invariant assertions (I-assertions), which are placed before calls to synch functions, consist of a union of global states. These states are the ones which are permissible when the process is executing in the code section preceding the call.

Result assertions (R-assertions), which are placed after synch function calls, specify the effects of the call upon the global state of the system and upon other processes, i.e., R-assertions indicate the impact of a contact between processes upon the entire system. However, the global state resulting from an R-assertion must be consistent with the following I-assertion as well as I-assertions for all other code sections executed concurrently. Hence, I-assertions restrict the impact of other R-assertions during the execution of asynchronous code.

When a SAL synchronization problem description, including definition of global states and abstract program skeletons for the process types, is input to the SYNVER system, SYNVER will synthesize code for the synch functions and then apply a verification technique to the specifications to prove them correct. The correctness of the code generated for the synch functions follows from the correctness of the synthesizer. This may be achieved using standard techniques and will not be discussed further. Here, we are concerned with the verification techniques which are applied to the specifications.

Verifying that a SAL specification is correct means proving that the invariant and result assertions are all mutually consistent; if (i) the system of processes is initially in a state which is consistent with the I-assertions for the initial code section for all processes, and (ii) there exists no sequence of synch function calls such that the system is put into a state which is inconsistent with the I-assertions of the code sections being evaluated concurrently, then no I-assertion will ever be violated. Since the I-assertions, in essence, describe the synchronization desired, the specifications have been shown to be correct.

5. THE SAL SPECIFICATION LANGUAGE

The first portion of a SAL problem specification is a set of state definitions of the form:

<name> IS <conjunction of Boolean relations on state variables>

for example,

NULL IS NREAD = 0 AND NWRITE = 0.

The rest of the SAL specification consists of one or more process type definitions of the form:

<process name> DOES <asynchronous code interspersed with synch function calls>

Synch function calls occur at the points of contact with other processes and are intended to provide the appropriate synchronization between processes. SAL assertions must be provided before and after each synch function call to describe the actions or the impact that the user expects of these synch functions. SAL provides two types of assertions: invariant assertions (I-assertions) and result assertions (R-assertions), which act as pre- and post-conditions, respectively. I-assertions are specified to hold for entire sections of asynchronous code (the section preceding the call), and they indicate what global states may hold upon the initiation of the call. R-assertions follow each call and indicate the effect on the entire set of processes of having performed that call. The effect of a call may be to change the global state of the system and to make processes wait or to cause previously waiting processes to continue.

When one proves entire (sequential) programs correct, an output assertion is associated with the halting point of the program. It is often characteristic of synchronization problems that there is no such halt and corresponding output assertion. Consequently, SAL assertions do not behave as input-output assertions for a synch function call. We emphasize that SAL assertions are not like Hoare assertions [Hoare69] in this respect. An R-assertion describes the state of the system after the evaluation of the associated synch function call, regardless of whether or not the calling process continues execution.

To facilitate the presentation of SAL syntax as well as the verification technique, we will introduce an example, the "first readers and writers problem" [Cour71]. There is a table of data, access to which is shared by all processes. Some processes are writers who require exclusive access to the table. Others are readers accessing the table on a read-only basis. Any number of readers may access the table at a time. Readers may not access the table while a writer has access, and vice versa. Neither type of process has priority when the table becomes available. We require two integer state variables:

NREAD, the number of readers accessing the table,
NWRITE, the number of writers accessing the table.

There are three states of the system:

NULL IS NREAD = 0 AND NWRITE = 0
WRITING IS NREAD = 0 AND NWRITE = 1
READING IS NREAD GT 0 AND NWRITE = 0

There are two types of processes, READERS and WRITERS. READERS perform the synch functions STARTREAD followed by ENDREAD, while WRITERS perform the synch functions STARTWRITE then ENDWRITE.

We can now discuss the syntax of invariant and result assertions. Invariant assertions immediately precede each synch function call and describe the state of the entire system of processes throughout the evaluation of the preceding code section, up to the start of the evaluation of the synch function call. The syntax of an invariant assertion is:

ASSERT(S1, ..., Sn)

where Sj's are names of states already defined by the SAL specification. The system may be in only one state at a time. Hence the state definitions are disjoint; exactly one of the Boolean expressions defining the states is true at a time. The meaning of an invariant assertion, then, is that throughout the asynchronous code preceding the synch function call the state of the system is a member of the specified set of states. In our example (see Appendix) the invariant assertion ASSERT(READING) holds throughout the reader's critical section, while the invariant assertion for the non-critical sections is ASSERT(NULL, WRITING, READING), i.e., all the states.

The syntax of result assertions is much richer than that of invariant assertions. We will give only a brief description of the syntax here. The reader is referred to [Grif75a, Grif75b] for greater detail and a more complete description of SAL. Result assertions are placed immediately after every synch function call. They specify the transitions between states of the system to be effected by the preceding synch function call. Result assertions consist of a series of conditionals separated by commas. The basic format of a result assertion is:

ASSERT(cond-1, cond-2, ..., cond-n)

where each cond-j is a conditional whose syntax is of the form:

possible current state → next state AND control actions

We call the possible current state on the left hand side of → the initial state of the conditional. Correspondingly, we call the next state on the right hand side the final state. The semantics of an individual conditional in a result assertion are that if the system state is the initial state, the synch function should change the system to the final state, performing the control actions specified on the right hand side of the conditional. The conditionals of a result assertion, ASSERT(cond-1,...,cond-n), are to be considered in order of appearance, as in a LISP conditional. If the state of the system is not the initial state of cond-1, then consider cond-2, etc. The union of the initial states of the conditionals of a result assertion is equal to the set of states listed in the preceding invariant assertion.

The control conjuncts WAIT and PROCEED on the right hand side of a result assertion conditional are possible control actions which may be specified by a result assertion. They indicate whether or not control is to be transferred back to the process performing the synch function call immediately after the call terminates. The conjunct PROCEED may be omitted; if WAIT is not specified, PROCEED is assumed.

In the last conditional of a result assertion, SAL permits the ELSE operator to replace the initial state and then "→" as syntactic sugar for all the remaining states of the preceding invariant assertion not already explicitly listed in any other conditional. In the case that there is only one conditional in the result assertion, ELSE may be replaced by ALWAYS for readability.

Boolean expressions involving the variables of the invariant states may appear on the left hand side of →. An example is:

READING AND (NREAD = 1) → NULL .

We now present a brief discussion of several other control conjuncts that may appear on the right hand side of a result assertion conditional. We define sets of processes, called waitsets, which contain those processes which are required to WAIT. A process may belong to at most one waitset at a time.

There are three control conjuncts on the right hand side of result assertion conditionals which describe waitsets and operations upon them.

WAIT IN <waitset name>

enters the process performing the synch function call as a member of the set <waitset name> and prevents it from continuing execution.

Corresponding to this is the conjunct STARTUP.

STARTUP [ALL] <process type> OUTOF <waitset name>

specifies that (all) processes(es) of type <process type> are to be removed from the waitset <waitset name> and scheduled for execution. Execution will begin with the next statement in the waiting process after the synch function call.

If the IN <waitset name> or OUTOF <waitset name> are omitted, the <waitset name> is assumed to be the same as the <process type> name.

The third control conjunct related to waitsets is

REVIVE [ALL] <process type> OUTOF <waitset name> .

REVIVE removes (all) process(es) of type <process type> from the waitset <waitset name> and allows them to redo their synch function. Note that REVIVE does not necessarily cause a process to continue execution beyond the synch function call, as the process may again be blocked.

The purpose of REVIVE is to make a clear distinction between the actions of the process performing the wakeup and the actions of the process being reawakened. If STARTUP is used to wake up a waiting process, the reawakened process will begin execution at the statement following its synch function call, thereby having no opportunity to make any changes in the system state or perform any control actions. Hence, any such actions must be performed by the process performing the STARTUP. In general, the use of STARTUP in this fashion would require distribution of a copy of a reawakened process' result assertion into every conditional where a process of that type might be reawakened. REVIVE achieves an equivalent effect in a more modular fashion, in that it in effect generates a call on the revived process's synch function as a subroutine. Thus, knowledge about the actions of a synch function is localized, and the entire specification becomes easier to write and more concise.

The REVIVEd result assertion may itself contain REVIVE conjuncts and nesting of REVIVEs may continue to arbitrary depth. In this case, or when a sequence of REVIVEs occurs on the right hand side of a conditional, a chain of interim states is created. Each successive REVIVE takes as its initial state the state resulting from the previous REVIVE.

For some states, several actions might be taken with different priorities. The conditional specifying this uses the TRY conjunct, or priority set. The priority set begins with the keyword TRY and consists of one TRY clause followed by an arbitrary number of NEXT clauses. Each of the TRY or NEXT clauses will be called a branch of the priority set. Each branch is considered in turn. The branches are listed in order of priority from highest to lowest.

The format of a priority set is:

```
lhs → right hand side THEN TRY IF actions-1 POSSIBLE
                           NEXT IF actions-2 POSSIBLE
                           ...
                           NEXT IF actions-n POSSIBLE
                           END-TRY .
```

Each branch consists of a set of control conjuncts at least one of which is enclosed by the words IF ... POSSIBLE. The IF ... POSSIBLE brackets enclose the control operations which may not necessarily be possible. The highest priority branch found to be possible is the one chosen. An example of the use of the priority set would be in the second readers and writers problem [Cour71], where when the table becomes available, we grant the use of it to a waiting writer, if any. If there are no waiting writers, then all waiting readers are granted the table. Thus writers have priority over readers.

It is sometimes the case that several alternatives for a result assertion conditional are desired with equal priority. An example is the first readers and writers problem [Cour71] described above. When the table becomes available, either one waiting writer or all waiting readers may be granted access, and we explicitly do not wish to favor either process type. The SAL conjunct which permits one to specify this is an OR conjunct. The format of an OR conjunct is:

```
EITHER actions-1 OR actions-2 OR ... OR actions-n END-OR .
```

Ordinarily all branches of an OR conjunct are assumed to be possible. When an OR conjunct is contained in a branch of a priority set, however, this is no longer true. Within a priority set, all the branches of an OR conjunct must be considered and rejected before the next branch of the priority set can be considered.

6. INFORMAL DESCRIPTION OF THE VERIFICATION

The verification that a given specification is correct is by induction. We will explain the proof technique using the first readers and writers problem as an example.

The SAL specification contains a list of the states of the system. It also provides a process type description with the order of synch function calls for each type of process which will be in the system of concurrent processes. Each synch function call is preceded by an invariant assertion and followed by a result assertion. Since a synch function is in essence defined by these assertions, if a synch function is used more than once, it must have the same assertions associated with it. For each process type definition, the verifier constructs an abstract program. For each synch function call, the abstract program contains four items of information:

(1) A code index number representing the asynchronous code preceding the synch function call.
(2) The set of states specified by the invariant assertion.
(3) The name of the synch function.
(4) The result assertion in an internal form.

We will use the names c_i, a_i, f_i, and r_i to refer to the above items, respectively. The verifier gives the code following the last synch function call in each process description the same index as the code preceding the first synch function call in that description. The purpose of this is to help simulate the effect of more than one process of each type running concurrently.

The first readers and writers problem has the following abstract programs. The complete SAL specification appears in the Appendix.

Reader: c_1

a_1: (NULL, WRITING, READING)

STARTREAD

r_1 /* result assertion for STARTREAD

c_2

a_2: (READING)

ENDREAD

r_2 /* result assertion for ENDREAD

c_1

Writer: c_3

a_3: (NULL, WRITING, READING)

STARTWRITE

r_3 /* result assertion for STARTWRITE

c_4

a_4: (WRITING)

ENDWRITE

r_4 /* result assertion for ENDWRITE

c_3

One may visualize the effect of synch function call as causing a transition from a code section c_i to zero or more result code sections depending upon whether the process PROCEEDs or WAITs and whether any other processes are REVIVEd or caused to STARTUP. Similarly these synch functions make transitions from one state of the system to one of a set of several possible result states depending upon the initial and final states of the conditionals of the corresponding result assertions.

We can represent these transitions among states and among code sections as a transition table with transitions between ordered pairs of the form:

(state, set of code sections)

The transition table for the first readers and writers problem is given below. If the system is currently performing the synch function f_i and the state of the system is s_j, then the resulting transition is one of the ones listed for the ordered pair (s_j, c_i). There may be several transitions possible depending on which branches of a priority set or OR conjunct are used, as well as the truth value of the Boolean expressions which may occur on the left hand side of a conditional.

Synch Function	Current Pair	Set of Result Pairs
STARTREAD	(NULL, c_1)	(READING, c_2)
	(READING, c_1)	(READING, c_2)
	(WRITING, c_1)	empty
ENDREAD	(READING, c_2)	(READING, c_1)
		(NULL, c_1)
		(WRITING, (c_1, c_4))
STARTWRITE	(NULL, c_3)	(WRITING, c_4)
	(WRITING, c_3)	empty
	(READING, c_3)	empty
ENDWRITE	(WRITING, c_4)	(WRITING, (c_3, c_4))
		(READING, (c_2, c_3))
		(NULL, c_3)

A transition table like this may be constructed mechanically from the result assertions corresponding to each synch function call. The transition table specifies how each synch function call changes the state of the system and what code sections control will enter as a result of each call. If waiting processes become unblocked as a result of a STARTUP or REVIVE specified by a call, then several code sections may be entered simultaneously. This is the case for ENDREAD when the system is in state READING and NREAD=1; then not only does the reader process continue (code c_1 is entered), but a writer is REVIVEd. Thus the entry in the transition table is (WRITING, (c_1, c_4)). In a later section we will provide an algorithm for constructing the transition table.

We will refer to the set of result pairs in the transition table which correspond to a given current ordered pair as the entry-set. Any individual ordered pair in the entry set will be called an entry. The first element of an entry will be referred to as entry.state, while the second will be referred to as entry.code-set.

In general when control is simultaneously in code sections $c_1,\ldots,c_n$, a state S of the system is allowable if and only if S is a member of the INTERSECTION ($a_1,\ldots a_n$), where the a_i's, $1\leq i\leq n$, are the sets of states listed in the invariant assertions corresponding to the code sections c_i. If this is not true, then an invariant assertion has been violated; i.e., control is in some c_j, but the state of the system S is not a member of a_j. It is required, therefore, that for each entry in each entry-set of the transition table, entry.state must be an allowable state for entry.code-set. This requirement can easily be checked mechanically. This will guarantee that the performance of a synch function call does not immediately change the state of the system to a state which is not allowable.

In order to show that no result assertions violate any invariant assertions, we must prove the following: For any situation where a synch function is called, the resultant state of the system does not violate the invariant assertion of any code section in which control resides. These code sections include the sections resulting from the transition as well as sections evaluated concurrently with the synch function call. Thus, the resultant state of the system must be allowable for the set of simultaneous code sections.

This is not the only criterion which must be satisfied however. Implicit in the assertions are bounds on the number of processes which may be executing a code section concurrently. We define a function B on the set of states such that B(S) returns a set of code sections which may be executed by no more than one process at a time while the system is in state S. For the first readers and writers problem, the function B is as follows:

$$B(\text{WRITING}) = c_4 ,$$

i.e., only one writer may access the table at a time.

We refer to the state of the system plus the set of code sections being evaluated as a system descriptor, which we will write (S, SCS), where S is the state and SCS is the set of simultaneous code sections. We define a system descriptor (S, SCS) to be legal if and only if:

(1) S is an allowable state for SCS. This also implies that the intersection of the a_i's corresponding to the c's in SCS is non-empty.
(2) If any c_i in B(S) is also in SCS, then no more than one process is evaluating the code section c_i.

In the next section we will present a better representation for system descriptors and describe the induction which will prove that a specification is correct.

7. FORMAL DESCRIPTION OF THE VERIFICATION

We now present the formal description of the induction proof necessary to prove that a SAL specification is correct. The proof technique is similar to computation induction [Man71]. This induction proves that if the system does not violate any invariant assertions initially, then there is no sequence of subsequent synch function calls which will violate any invariant assertions. In this section we describe the induction formally and indicate how it also proves that a specification is deadlock-free, i.e., total deadlock as defined by Holt [Holt71a] [Holt71b], where all processes are blocked. We do not detect, except by inspection in individual cases, whether a given process becomes deadlocked.

We first make a number of definitions.

Let C be the set of all code sections c_i in a specification. Let NC denote the cardinality of C, the total number of code sections. Let c denote any arbitrary member of C.

Let S be the set of all states of the system. Let s denote any arbitrary member of S.

Let $M:C\rightarrow 2^S$ be a function defined on any code section, whose range is an element of the power set of S, such that

$$M(c_i) = a_i$$

M returns the set of states specified by the invariant assertion associated with the code section c_i.

Let N denote the set of natural numbers {0,1,...}.

Let F denote the set of functions f such that f:C→N. Then f(c) is a non-negative integer for any f and any c. f(c) will be used to count the number of processes evaluating c.

Let DELTA be the set S×F. Let <s,f> denote an arbitrary member of DELTA. We call an element of DELTA a system descriptor.

As noted in the previous section, there are certain code sections which may be evaluated by no more than one process at a time. These restrictions depend on the state of the system. We define a function

$$B:S \rightarrow 2^C$$

such that for a given s, B(s) is the set of all code sections which no more than one process at a time may evaluate when the system is in state s.

We define a set D which is a subset of DELTA such that

<s,f> is in D if and only if
1) s is in INTERSECTION(M(c))
f(c)>0
2) c in B(s) IMPLIES f(c) ≤ 1

This set D is a representation of all the legal system descriptors.

The Base Step

The base step of the induction can now be described. Let INITIAL be the subset of DELTA which consists of all the possible initial system descriptors (i.e., the configurations before any synch function calls are made). Then the base step of the induction is simply to show that

INITIAL is a subset of D.

The Induction Step

For each c in C we define a function $gamma_c$ which describes the actions made by the synch function call following code section c. For every c in C,

$$gamma_c : D \rightarrow 2^{DELTA}$$

where $gamma_c$ (<s,f>) is empty if and only if f(c)=0. So $gamma_c$ is totally defined on D. As explained in the previous section, a synch function call for a given state s may result in different actions depending on which branch of a TRY or EITHER clause was performed, etc. So $gamma_c$ (<s,f>) returns an element of 2^{DELTA}. This set is always finite, however, because the number of conditionals in a result assertion and the number of branches of TRY and EITHER clauses are always finite.

Now we can define a function GAMMA

$$GAMMA:D \rightarrow 2^{DELTA}$$

such that

$$GAMMA(<s,f>) = \underset{c \text{ in } C}{UNION}(gamma_c(<s,f>))$$

For a given <s,f>, GAMMA(<s,f>) is the set of all possible system descriptors which might result from the occurrence of any synch function call that could be performed at that point in the computation.

The induction step is to show that

<s,f> in D IMPLIES GAMMA(<s,f>) is a subset of D.

We define <s,f> to be a descendant of $<s,f>_0$ if <s,f> = $<s,f>_0$ or <s,f> is in GAMMA(<s,f>') for some descendant <s,f>' of $<s,f>_0$.

If the base step and the induction step are proven, then we may infer that if <s,f> is in INITIAL then every descendant <s,f>' of <s,f> is in D. Hence, from any initial configuration of the system, there is no sequence of synch function calls which causes any invariant assertion to be violated.

D is constructed from the invariant assertions of the specification together with information about how many processes may be executing a code section at a time, i.e., the definition of the function B. The function GAMMA is constructed from the individual $gamma_c$'s which are constructed from the result assertions.

8. THE IMPLEMENTATION

It remains now to show how the sets INITIAL and D can be obtained, and how to construct each function $gamma_c$, the function M, and the function B.

INITIAL can be obtained from the set of states S and the abstract programs for each process type in the system. At system initialization, the initial state of the system is the null or default state where no resources are in use. Usually this state is named NULL. Then for each process type p some number N_p of processes of type p are started up. By inspection of the abstract program for processes of type p, it can be determined which code section c_p is the first code section for processes of type p. Let Z denote the set for all such c_p.

Then INITIAL = $\{<NULL,f> \mid f(c_p)=N_p$ for all c_p in Z
$f(c)=0$ for c in C-Z$\}$.

For the moment, let us assume that the function B is supplied by the user as additional information. Later we will indicate how the SYNVER verification system can help with the construction of B.

The definition of the function M is clear. For each code section c_i, $M(c_i)$ returns a_i, the set of states specified by the invariant assertion which holds for c_i.

From these sets and functions D can be constructed. Actually D is almost always an infinite set, even with the restrictions on f made by B, because for c's not contained in B(s) for any s, f(c) can be any non-negative integer. Thus there are an infinite set of such f's for each unrestricted c. Consequently the set D cannot be enumerated in a finite number of steps. If D were finite, then we could take each element <s,f> in D and check if GAMMA(<s,f>) is a subset of D. Then the induction step of the verification would be proven and would terminate in a finite number of steps.

By the construction of the set D and the definition of each $gamma_c$, it is clear that the relevant values of any f(c) are 0, 1, and 2 or more:

1) f(c)=0 -- This corresponds to no processes evaluating code section c.
2) f(c)=1 -- This corresponds to exactly one process evaluating code section c.
3) f(c) > 1 -- This corresponds to two or more processes evaluating code section c.

This justifies the definition of three equivalence classes of the values of f(c); for each c in C we define the relation $EQUIV_c$ as follows:

$$<s,f> \ EQUIV_c \ <s',f'> \text{ if and only if } s=s' \text{ and } G(f(c)) = G(f'(c))$$

where $G:N \to N$ is a function such that

$$G(0) = 0$$
$$G(1) = 1$$
$$G(n) = 2 \text{ for } n \geq 2.$$

We can now define the equivalence relation EQUIV such that

$$<s,f> \ EQUIV \ <s',f'> \text{ if and only if for all c in C, } <s,f> \ EQUIV_c \ <s',f'>$$

With this equivalence relation, D is partitioned into a finite number of elements.

Thus the induction step of the verification can be proven by examining only the elements of the partition of D induced by EQUIV. There are at most 3*NC*NS such elements of the partition where NC is the cardinality of C and NS is the cardinality of S. This is a finite set, and hence the induction step of the verification can be proven in a finite number of steps.

With this technique deadlock can also easily be detected. By the definition of D, there are no elements <s,f> of D such that f is identically zero. This is because <s,f> in D implies s in $\text{INTERSECTION}_{f(c)>0}(M(c))$, which is empty if f(c)=0 for every c. Consequently, if every descendant of any element of D is also in D, the solution is free from total deadlock. In order to compute GAMMA(<s,f>) for any <s,f> in D, we need to compute $gamma_c$(<s,f>) for each c in C. $gamma_c$ is determined from the transition table which is constructed from the result assertions of the specification. We now present the algorithm for constructing the transition table. Then we will describe how to construct each $gamma_c$ from the table.

The Transition Table

The transition table describes for each code section c in C and for each state s in S, how the synch function call following code section c affects both the state of the system and the set of simultaneous code sections. The transition table is constructed from the result assertions of the SAL specification together with the abstract program produced for each process type.

An abstract program is constructed for each process type from the SAL specification. Every piece of code of the form:

```
<asynchronous code>
ASSERT(---);
synch function call;
ASSERT(---);
```

is abstracted to a code index c_i, a set of states for the invariant assertion a_i, a synch function f_i and a result assertion r_i.

The entries in the transition table are formed from examination of the conditionals of each result assertion.

One row in the table is constructed for each (s,c_i) pair, where s ranges over all the states in the corresponding a_i. For every state s in a_i, the result column of the transition table represents all the actions specified by all the conditionals whose initial state is s in the result assertion r_i. Sometimes there are several such conditionals in r_i, each with different Boolean expressions. Thus there may be multiple entries for each (s,c_i) line in the table. In addition, each branch of a TRY or EITHER clause causes another entry to the (s,c_i) line in the table.

For example,

```
s1 → s2 THEN TRY B1
             NEXT B2
               .
               .
               .
             NEXT Bn
             END-TRY
```

creates n entries in the transition table, one for each of the branches B_j. EITHER constructs are treated similarly.

Each individual result pair in the entry-set consists of two parts:

(1) A state which the system may be in after the completion of the synch function call.
(2) A list of code sections where control will be as a result of the synch function call. This is empty when the process performing the synch function call must WAIT, and no other processes are REVIVEd or caused to STARTUP.

Although the system is actually in only one state at a time, it may not be decidable when constructing the transition table which state this is. For example, for code section C1 if the result assertion is:

```
ASSERT(S1 and K=1 → S2,
       S1 and K=2 → S3) ,
```

the (S1,C1) part of the transition table has two entries, (S2, C2) and (S3, C2). Actually the system will be either in state S2 or in state S3, depending upon the value of K when the synch function call is made. Statically, however, the only information that can be ascertained is the entries (S2, C2) and (S3, C2). So, if for a given c_i the result assertion r_i has multiple conditionals for a given state s, then the (s,c_i) line of the table consists of one entry for each conditional with initial state s.

Let us now consider how to construct an individual entry for a given c_i and s, and a given set of branches of TRY and EITHER clauses. This narrows the set of conjuncts to be considered on the rhs of a conditional to be rhs state, STARTUP, REVIVE, WAIT, and PROCEED. Initially, entry.code-set and entry.state are both empty.

If the conditional PROCEEDs (i.e., does not WAIT), then the next code index in the list of abstract program code (described above) for that process should be included in entry.code-set. Let us refer to this index as c_j. In most cases j=i+1.

If the conditional WAITs, entry.code-set remains unchanged.

If there are no REVIVE conjuncts on the right hand side of the conditional, then entry.state is the rhs state of the conditional.

For example, if the conditional is

$$s \rightarrow s_n \text{ AND PROCEED},$$

the entry is (s_n, c_j), where c_j is the next code section index in the abstract program.

If one or more STARTUP conjuncts appear on the right hand side of the conditional, one code index for each STARTUP conjunct is added to entry.code-set. This is the code index which follows the synch function call which caused a process of that type to WAIT.

The remaining conjunct to consider is REVIVE. This affects both parts of a transition table entry. Recall that a REVIVEd process redoes the synch function which caused it to WAIT. The REVIVEd process takes as its initial state the final state of the conditional specifying REVIVE. Consequently, the state of the system upon completion of the synch function call may not necessarily be that final state. In addition, if the REVIVEd process eventually PROCEEDs, the code section following the synch function call (which caused the REVIVEd process to wait) should be added to entry.code-set.

To construct an entry for (s,c_i) when one of the conditional's conjuncts is a REVIVE, one first constructs the entry for that conditional without considering any REVIVEs. This results in a set of one or more result pair entries. Then, for each such entry E, apply the following procedure to the synch function of the REVIVEd process:

> Let c_j denote the code section which corresponds to the synch function of the REVIVEd process.
> Replace E by the new-entries obtained by:
> for each entry E' from the result pairs corresponding to (E.state,cj)
> (1) Let new-entry.state be E'.state
> (2) Let new-entry.code-set be UNION(E.code-set,E'.code-set)

This results in some new set of (s,c_i) entries. If there is another REVIVE in the (s,c_i) conditional above, then this procedure is applied to each entry E in this new set of (s,c_i) entries for the synch function which caused this next REVIVEd process to WAIT.

In general, a REVIVEd process may in turn REVIVE another process, and so on. This creates a transitive closure problem of the chain of REVIVE references to (s,c_i) entries. If this chain of references has a minimal element (one which does not make a further REVIVE reference) then the corresponding transition table entries can be completed. Otherwise the chain of references is strongly connected. Tarjan [Tarj72] describes an algorithm which detects strong-connectivity. If the chain of REVIVEs is strongly connected, then this means a process eventually tries to REVIVE itself or a process identical to itself. In these cases another method must be found for completing the transition table, after which this verification technique may be applied.

Once the transition table entries have been completed by performing the transitive closure of all chains of REVIVEs, we can construct the functions $gamma_c$.

The construction of the transition table from the result assertions is completely mechanizable with the exception of completing the transition table in the case of a strongly connected chain of REVIVEs.

Constructing the Function GAMMA

Once the transition table is completed, it is easy to construct each $gamma_c$, and therefore the function GAMMA as well. The transition table contains entries for each c in C and each s in M(c). The transition table may have several entries for each such c and s, and each entry has two parts, entry.code-set and entry.state. Then for each c, and each <s,f> in D, $gamma_c$ (<s,f>) is empty if f(c) = 0. Otherwise (f(c)#0),

$$gamma_c(<s,f>) = \{<E.state, f_E> \mid \text{for all entries E in the transition table for } (s,c)\}$$

where f_E is the function in F such that

(1) $f_E(c) = f(c)-1$, and

(2) for each c_i in E.code-set $f_E(c_i) = f(c_i) + 1$.

EXAMPLE: THE FIRST READERS AND WRITERS PROBLEM

S = {NULL,WRITING,READING}
C = {c_1, c_2, c_3, c_4}

Synch Function	Current Pair	Set of Result Pairs
STARTREAD	(NULL, c_1)	(READING, c_2)
	(READING, c_1)	READING, c_2)
	(WRITING, c_1)	empty
ENDREAD	(READING, c_2)	(READING, c_1)
		(NULL, c_1)
		(WRITING, (c_1, c_4))
STARTWRITE	(NULL, c_3)	(WRITING, c_4)
	(WRITING, c_3)	empty
	(READING, c_3)	empty
ENDWRITE	(WRITING, c_4)	(WRITING, (c_3, c_4))
		(READING, (c_2, c_3))
		(NULL, c_3)

The function B supplied to the verifier is as follows:

B(READING AND NREAD=1) = {c_2}

B(WRITING) = {c_4}

Elements of the Partition of D

		$f(c_1)$	$f(c_2)$	$f(c_3)$	$f(c_4)$	
d1:	<NULL,	GE 1	0	0	0	>
d2:	<NULL,	0	0	GE 1	0	>
d3:	<NULL,	GE 1	0	GE 1	0	>
d4:	<READING,	0	1	0	0	>
d5:	<READING,	0	1	GE 1	0	>
d6:	<READING,	GE 1	1	0	0	>
d7:	<READING,	GE 1	1	GE 1	0	>
d8:	<READING,	0	GE 2	0	0	>
d9:	<READING,	GE 1	GE 2	0	0	>
d10:	<READING,	0	GE 2	GE 1	0	>
d11:	<READING,	GE 1	GE 2	GE 1	0	>
d12:	<WRITING,	0	0	0	1	>
d13:	<WRITING,	GE 1	0	0	1	>
d14:	<WRITING,	0	0	GE 1	1	>
d15:	<WRITING,	GE 1	0	GE 1	1	>

Notice that we have collapsed this partition for c_1 and c_3, because the classes =1 and GE 2 behave equivalently.

Base Step

INITIAL = {<NULL,	GE 1	0	0	0 >
<NULL,	0	0	GE 1	0 >
<NULL,	GE 1	0	GE 1	0 >}

Clearly INITIAL is a subset of D.

Induction Step

GAMMA(d3) = {d5, d7, d13, d15}
GAMMA(d7) = {d3, d6, d7, d10, d11, d15}
GAMMA(d11) = {d7, d9, d11}
GAMMA(d15) = {d3, d7, d11, d13, d14, d15} .

We have taken advantage of an optimization here which is described in the following section. Therefore it was not necessary to apply GAMMA to a representative element of each equivalence class in the partition of D.

9. EXTENSIONS

The verification technique presented above is easily mechanized as it consists of iterating through an enumeration of a finite set, the partition of D, and applying the function GAMMA to each representative element of that partition.

In addition, the function B may be generalized. In most synchronization problems either an unbounded number of processes are permitted to evaluate a critical section simultaneously or at most one process may be evaluating it. A problem could be formulated, however, in which at most some bounded number (greater than 1), say N, processes are permitted to evaluate a critical section simultaneously. Then B would have to be generalized such that c in B(s) IMPLIES $f(c) \leq N$. The verification technique can accommodate this generalization of B without difficulty. Since N is finite, the number of equivalence classes in the partition of D is also finite, and thus the verification may be accomplished as above.

This verification technique can be optimized to avoid having to apply GAMMA to every representative element of the partition of D. Let us define a "less than" relation, <, on functions such that

$f' < f$ if and only if there exists a c_i such that $f'(c_i) = 0$, $f(c_i) > 0$ and $f(c) = f'(c)$ for $c_i \neq c$.

Then clearly $f' < f$ implies GAMMA(<s,f'>) is a subset of GAMMA(<s,f>) since

$$\bigcup_{c \text{ in } C} \text{gamma}_c(\langle s,f' \rangle) \text{ is a subset of } \bigcup_{c \text{ in } C} \text{gamma}_c(\langle s,f \rangle) .$$

Thus for any two representative elements of the partition <s,f> and <s,f'>, if f'<f, then only <s,f> need be considered.

10. CONCLUSION

The verification technique presented here is based on the existence of synch functions to achieve synchronization among concurrent processes. Synch functions are separate from the asynchronous code of a program. This simplifies the correctness proofs of both the synchronization and the asynchronous code because the proofs can be done independently. Consequently, the combinatoric explosion of program paths to be proven as in the Levitt-Manna-Aschcroft approach is reduced. This is achieved without making assumptions that some bodies of code are protected. Furthermore, our technique is easily mechanizable and is suitable for incorporation in an automatic programming system. In addition, we have several implementations for synch functions, including Prenner's control interpreter, Hoare's monitors, and Dijkstra's P and V operations. This flexibility is achieved because both SAL and the verification technique we use are at a sufficiently high level so as to be independent of the implementation.

APPENDIX

The following is the SAL specification of the first readers and writers problem [Cour71]:

```
NULL IS NREAD = 0 AND NWRITE = 0
WRITING IS NREAD = 0 AND NWRITE = 1
READING IS NREAD GT 0 AND NWRITE = 0

READER DOES
     BEGIN
          ASSERT(NULL,WRITING,READING);
          STARTREAD;
          ASSERT(<READING,NULL> →
                 <READING,READING>,
                 ELSE WAIT);
          <code for reading>
          ASSERT(READING);
          ENDREAD;
          ASSERT(READING AND NREAD GT 1 → READING,
                 if not last reader, just continue
                 READING AND NREAD = 1 →
                    NULL THEN
                       TRY IF REVIVE(WRITER) POSSIBLE
                         END-TRY);
                       if last reader, revive
                         waiting writer if any
          END;

  WRITER DOES
     BEGIN
          ASSERT(NULL,WRITING,READING);
          STARTWRITE;
          ASSERT(NULL→WRITING, ELSE WAIT);
          if table free then go, else wait
          <code for writing>
          ASSERT(WRITING);
          ENDWRITE;
          ASSERT(ALWAYS NULL THEN
                    TRY EITHER IF REVIVE(WRITER)
                         POSSIBLE OR
                         IF REVIVE(ALL READER)
                           POSSIBLE END-OR END-TRY);
                    exit, letting writer or all readers go if any
     END;
```

REFERENCES

[Ash70] Ashcroft, E. and Manna, Z., Formalization of Properties of Parallel Programs. Machine Intelligence Vol. 6, 1970.

[Ash75] Ashcroft, E. A, Proving Assertions about Parallel Programs, J. Comp. and Sys. Sci. 10, pp 110-135 (1975).

[Byrn74] Byrn, W. H., "Sequential Processes, Deadlocks, and Semaphore Primatives," Ph.D. Thesis, Harvard University, August 1974.

[Cour71] Courtois, P. J. et al., "Concurrent Control with 'Readers' and 'Writers'," Comm. ACM Vol. 14, No. 10 (October 1971), pp. 667-668.

[Dij68] Dijkstra, E. W. "Cooperating Sequential Processes," in Programming Languages, (F. Genuys, ed.), Academic Press, New York, 1968, pp. 43-112.

[Floyd67] Floyd, R. W. "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science (J. T. Schwartz, ed.) Vol. 19, American Math. Soc., Providence, Rhode Island, 1967.

[Grif74] Griffiths, P., SYNVER: A System for the Automatic Synthesis and Verification of Synchronous Processes, Proceedings ACM74, Nov. 1974.

[Grif75a] Griffiths, P., SYNVER: An Automatic System for the Synthesis and Verification of Synchronous Processes, Ph.D. Thesis, Harvard University, June 1975.

[Grif75b] Griffiths, P., SAL: A Very High Level Specification Language, Proceedings International Symposium on Proving and Improving Programs, July 1975.

[Hoare69] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, Comm. ACM Vol. 12, No. 10 (October 1969).

[Hoare74] Hoare, C. A. R., Monitors: An Operating System Structuring Concept., Comm. ACM Vol. 17, No. 10 (October 1974), pp. 549-557.

[Holt71a] Holt, R. C., On Deadlock in Computer Systems, Ph.D. Thesis, Cornell University, January 1971.

[Holt71b] Holt, R. C., Some Deadlock Properties of Computer Systems, Symp. on Opg. Sys. Principles, Palo Alto, October 1971.

[Lev72] Levitt, Karl N., The Application of Program-Proving Techniques to the Verification of Synchronization Processes, Fall Joint Computer Conference, pp. 33-47, 1972.

[Man69] Manna, Z., The Correctness of Programs, Journal of Computer and System Sciences, 3, 119-127, 1969.

[Man70] Manna, Z., The Correctness of Nondeterministic Programs, Artificial Intelligence I, pp. 1-25, 1970.

[Man71] Manna, Z., Ness, S., and Vuillemin, J., Inductive Methods for Proving Properties of Programs, Computer Science Department, Stanford University, 1971.

[Naur66] Naur, P., Proof of Algorithms by General Snapshots, BIT 6,4, pp. 310-316, 1966.

[Par75] Parnas, D. L., On a Solution to the Cigarette Smoker's Problem (without conditional statements), Comm . ACM Vol. 18, No. 3 (March 1975), pp. 181-183.

[Pren72] Prenner, C. J., Multi-path Control Structures for Programming Languages, Ph.D. Thesis, Harvard University, May 1972.

[Pren73] Prenner, C. J., Extensible Control Structures, SIGPLAN Notices, Vol. 8, No. 9, pp. 129-132, September 1973.

[Rob74] Robinson, L. and Holt, R. C., Formal Specifications for Solutions to Synchronization Problems, Computer Science Group, SRI, Menlo Park, California.

[Steele75] Steele, G. L., Generation of Optimized Semaphore Synchronization Code, Senior Thesis, Harvard University, May 1975.

[Tarj72] Tarjan, R., Depth-first Search and Linear Graph Algorithms, SIAM J. Comput. Vol. 1, No. 2, June 1972.

[Weg70] Wegbreit, B., Studies in Extensible Programming Languages, Ph.D. Thesis, Harvard University, May 1970.

[Weg74] Wegbreit, B. et al., ECL Programmer's Manual, Ctr. for Research in Comp. Tech., Harvard University, TR 23-74.

[Wulf74] Wulf, W. et al., HYDRA: The Kernal of a Multiprocessor Operating System, Comm. ACM Vol. 17, No. 6 (June 1974) pp. 337-344.

[Zilles73] Zilles, S., Procedural Encapsulation: A Linguistic Protection Technique, SIGPLAN Notices, Vol. 8, No. 9 (September 1973), pp. 142-146.